O comparaţie teoretică şi experimentală a unor metode de sortare

Costică - Vlad Luchian
Departamentul de Informatică.
Facultatea de Matematică şi Informatică
Universitatea de Vest din Timișoara, Timişoara
Email:costica.luchian98@e-uvt.ro

5 mai 2018

Rezumat

Imaginează-ți cât de greu ar fi să cauți cuvinte într-un dicționar dacă acestea nu ar fi aranjate alfabetic.

Sortarea este o problemă trivială în ziua de astăzi.
Însă aceasta nu a fost mereu așa. Problema sortării a apărut în momentul în care omul a început să manevreze obiecte puține, iar mai apoi s-a extins la un volum mare de obiecte. În informatică, aceste obiecte reprezintă date.

Fiind scopul acestei ramuri a științei de a mecaniza lucrul și eventual de a economisi timp, a apărut nevoia de introducere a unor mecanisme(algoritmi), care prin instrucțiuni clare să efectueze o rearanjare a obiectelor date după un anumit criteriu.

Ce este o sortare? Cum arată o sortare,ce algoritmi există și care este cel optim? Ce complexitate are un algoritm optim dar și cum arată o comparare a performanțelor fiecărui algoritm, toate aceste aspecte vor fi discutate pe parcursul acestei lucrări, care încearcă a face o comparație între cei mai uzuali algoritmi de sortare.

Cuprins

1 Introducere

Ce este un algoritm? Un algoritm este un sistem de reguli,pași urmați mecanic pentru a rezolva o problemă. Un algoritm de **sortare** este un algoritm care are drept scop (re)aranjarea unor elemente după un anumit criteriu.

Astfel, problema sortării este o problemă trivială, însă pentru o înțelegere mai aprofundată a uneltelor folosite pentru rezolvarea acesteia, o comparație a eficienței algoritmilor în detaliu urmează a fi facută. Un criteriu important în baza căruia se face clasificarea algoritmilor este **timpul de execuție** dar și **cantitatea de memorie necesară**. Algoritmii apăruți la început, erau simpli. Metoda Bulelor, metoda sortării prin Inserție dar și metoda sortării prin Selecție. Aceștia însă, nu sunt cei mai buni când vine vorba de manipularea unei cantități mare de date, având o complexitate de $\theta(n^2)$.[1].

Problema: Sortarea
Input: O secvență de n date-

$$d_1, d_2, ...d_n.$$

Output: Rearanjarea datelor, astfel încât

$$d_1 \leq d_2 ... \leq d_{n-1} \leq d_n.$$

Un exemplu de problemă în care se cere sortarea ar fi să sortăm efectiv o listă de note:

10,8,8,7,9,9,10,8,5,6,4.

Astfel, cel mai simplu algoritm de sortare *Bubble sort* va compara primul element, cu restul elementelor din dreapta sa, mutând-ul de fiecare dată o poziție, aducând-ul în poziția sa finală; apoi urmează al doilea element, și așa mai departe, până lista de note este sortată.
Fiecare algoritm de sortare promite un anumit tip de complexitate, fie $O(log(n)), O(n)$, $O(n+k)$, $O(nlog(n))$, $O(n^2)$. În dependență de cantitatea de date procesată, starea acesteia(dacă se știe) de a fi aleatorie/aproape sortată/sortată invers, se alege un algoritm anume.

[1]Pentru mai multe detalii despre analiza complexității algoritmilor, vezi [Ski] sect.2

Însă problemele în Informatică nu țin doar de sortări de zeci sau sute de elemente, ci țin de manipularea unei cantități mari de informații. De aceea aranjarea unei liste de sute de milioane de elemente, care pot fi numere sau cuvinte, fiind o problemă complicată, necesită algoritmi eficienți ca timp și memorie. Un exemplu pentru a demonstra cât de folosite sunt metodele de sortare, este orice căutare pe motoarele de căutare(ex. Google, Bing, Yahoo etc.). Astfel, pe lângă algoritmii de analiză complexi, la bază aceștia tot folosesc algoritmi de sortare, pentru a afișa articole cât mai relevante unei căutări.

Comparația:
Parametrii după care fiecare algoritm existent se etichetează a fi (in)eficient, sunt complexitatea timpului de execuție(timpul necesar pentru rularea programului), cantitatea de memorie necesară, dar și alți factori.

1.1 Scopul și obiectivul lucrării

Lucrarea *Compararea teoretică și experimentală a unor metode de sortare* are drept scop studiul și analiza printr-o comparație teoretică și experimentală a metodelor de sortare, în special: Bubble sort, Insertion sort, Selection sort, QuickSort[2], MergeSort[3] și CountingSort[4]. Într-u studierea aspectelor teoretice, am prezentat o scurtă descriere a fiecărei sortări, principiile, ordinul de complexitate. Pentru partea experimentală, am executat funcții sortare în Python, de 10^n elemente, pentru $n = \overline{1,4}$ (pentru cele de complexitate n^2) și pentru $n = \overline{1,7}$ pentru cele care am putut scoate un timp favorabil($\leq$ 10 minute). Pentru ca rezultatele să fie cât mai acurate, am repetat experimentul de 10 ori pentru fiecare metodă și număr de elemente. Elementele au fost generate aleator în același program, cu valori cuprinse de la 1 la numărul de valori al tabloului. Rezultatele experimentelor aferente sunt reprezentate în grafice. Alte scopuri ale acestei lucrări sunt: să exploreze, să verifice, să valideze, să confirme/infirme așteptările teoretice printr-un studiu empiric.

[2]QuickSort(sortarea rapidă) este o metodă de sortare dezvoltată de Tony Hoare și publicată în 1961 în jurnalul *Communications of the ACM*[Hoa]

[3]MergeSort(Sortare prin interclasare) este o metodă de sortare inventată de John Von Neumann și publicată în 1945; a fost publicat sub forma unui articol în jurnalul *IEEE Annals of the History of Computing*, vol. 15, no.1.[Neu]

[4]Counting sort este un algoritm care presupune că datele de intrare sunt numere $1..k$, k fiind cel mai mare număr din acea listă. **Acest algoritm funcționează doar când datele sunt numere.**

1.2 Planul lucrării

Această lucrare este formată din 5 capitole astfel:

- *1.Introducerea* propune o scurtă prezentare a problemei. De asemenea, este prezentă și descrierea unui exemplu trivial, pe lângă scopul și obiectivele lucrării lucrării.
- Secțiunea *2.Prezentarea formală a problemei* prezintă descrierea problemei și a soluției cu proprietățile sale, dar și ilustrează un exemplu.
- Capitolul *3.Implementare* propune un manual de utilizare a lucrării.
- *4.Studiu de caz* oferă date detaliate despre experimentul propus, datele de test, dar și explică și analizează rezultatele obțiunte.
- În secțiunea *5.Comparație cu literatura* sunt prezente analogii și comparații cu alte lucrări relevante în domeniu ce tratează aceeași problemă.
- Iar capitolul *6.Concluzii și direcții viitoare* formează un sumar a rezultatelor obținute dar și problemele deschisie rămase pentru o ulterioară rezolvare.

Limitări

Având în vedere că există o mulțime de algoritmi care au drept scop să sorteze obiecte, în această lucrare am studiat doar cei mai uzuali precum *BubbleSort, Insertion Sort, Selection Sort, QuickSort, MergeSort, Counting Sort.*

2 Prezentare formală a problemei

În această secțiune vom folosi notațile și formulele matematice din [Zah]. Deci, cum am menționat deja, sortarea unei cantități mari de obiecte (ea fiind, să zicem, de 10 milioane), este o problemă care necesită un algoritm cât mai optim.

Algoritmii, în care un pas arbitrar i, unde:

$$a_j \leq a_i \leq a_k, \forall j, i, k = \overline{1, n}$$

$$j \leq i \leq k.$$

este adevărat, sunt Bubble Sort, Selection Sort, Insertion Sort.[4] În general, putem asuma ca dupa $i-1$ pași, primele $i-1$ elemente sunt deja sortate. La momentul dat avem A[1..$i-1$] sortat, iar A[$i-1$..n] urmează a fi sortat. Timpul de execuție [5] al algoritmului este $T(n)$ unde n reprezintă dimensiunea tabloului sau numărul de elemente ce trebuie sortate. Pentru algoritmul de **Selection Sort** avem:

$$T(n) = \begin{cases} \theta(n), & x[1] \leq x[2] \leq ... \leq x[n] \\ O(n^2), & x[1] \geq x[2] \geq ... \geq x[n]. \end{cases}$$

Pentru **Selection** dar și **Bubble Sort**, timpul de execuție este

$$T(n) = \theta(n^2)$$

din cauza numărului mare de comparări executate, acesta fiind de n^2.

Însă algoritmilor care au la bază conceptul de *Divide et Impera*[6], precum *Quick Sort* sau *Merge Sort*, nu li se mai aplică aceeași regulă. Aici pentru *Merge Sort*, formula pentru *Merge Sort* pentru timpul de execuție este:

$$T(n) = \begin{cases} \theta(1) & n = 1. \\ 2T(n/2) + \theta(n) & n > 1. \end{cases}$$ [7]

cu formula generală:

$$T(n) = \begin{cases} T_0 & n \leq n_c. \\ kT(n/m) + T_{DC}(n) & n > n_c. \end{cases}$$

[4] O metodă care folosește acest algoritm este Shell sort.

[5] Timpul de execuție este timpul necesar pentru execuția tuturor prelucrărilor specificate în algoritm.

[6] Principiul Divide Et Impera presupune împărțirea problemei în subprobleme mai mici și rezolvarea recursivă a acestora. Dacă dimensiunea subproblemei este rezonabil de mică, rezolvarea acesteia se face într-o maniera simplă. La final are loc combinarea soluțiilor subproblemelor într-u obținerea soluția problemei originale.

[7] Pentru mai mult despre teorema Masters, vezi [Zah] sec 6.2 .

unde problema este de dimensiunea n, este descompusă în m subprobleme de n/m, dintre care se rezolvă k, $k \leq m$; T_{DC} este costul divizării și compunerii rezultatelor.

3 Implementare

Structurile care conțin datele pot fi referite ca tablouri sau liste. În lucrare, datele reprezentate sub formă de tablouri, au proprietatea de a avea indici, iar accesarea unui element se face cu ajutor indicilor.
În industrie, datele pot fi stocate sub forma unor structuri de date abstracte. Pentru mai multe detalii despre acestea, vezi SDA.

Problema sortării are drept date de intrare obiecte, fie numere, fie șiruri de caractere(nume, prenume, etc.), cheie în baza căreia se face sortarea. Soluția problemei este aceleași mulțime de obiecte, însă aranjate după un anumit criteriu.

Principiile de funcționare a algoritmilor prezentați:

- *Bubble sort* - începe de la primul element al tabloului și interschimbă repetat elemente alăturate care nu sunt în ordine.
- *Selection sort* - tabloul este "împărțit" în două subtablouri; algoritmul găsește valoarea minimă din tabel, o interschimbă cu prima poziție, și repetă acest pas până la finalizarea listei. O altă metodă mai eficientă care folosește același principiu este Shaker Sort, care în aceeași iterație plasează minimul la începutul tabloului și maximul la sfârșitul ei.
- *Insertion sort* - începem cu o "parte stângă" liberă. Valorile preluate din input sunt așezate într-un tablou. Astfel preluăm câte un element din tablou și o aranjăm în poziția corectă în partea I a tabloului. Pentru a găsi poziția corectă a elementului, comparăm valoarea cu cele deja existente în tabel, de la stânga la dreapta. Mereu,elementele din partea stângă a tabloului sunt sortate, iar cele din partea dreaptă nu.
- *MergeSort* - sortarea utilizând principiul "Divide et Impera" menționat mai sus.
- *QuickSort* - "Divide et Impera", paradigmă explicată anterior.[8].

[0]Precizare: Pentru mai multe detalii ce țin de formulele matematice, demonstrații dar si analiză mai detaliată, veziCORMEN Secțiunea 2.

[1]Pentru o evaluare a resurselor folosite, a datelor de intrare și ieșire, a timpilor de execuție, se va consulta link-ul următor.

[8]*Observație: Acest algoritm are nevoie și de un spațiu de memorie O(log n)

- *Counting Sort* - se parcurge tabloul pentru găsirea valorii maxime. Apoi se crează un tablou de k(valoarea maximă) valori, iar la parcurgerea tabloului original, la fiecare valoare din acesta, se incrementează valorea indexului potrivit din cel de-al doilea tabel. După finalizare se întoarce al doilea tablou, care conține elemente cu valori diferite de 0.

Un aspect ce ține de implentarea algoritmului de sortare *Quick Sort* este alegerea diferită a pivotului în diferite ipostaze a aranjării valorilor în tablou. Astfel pentru un tablou de elemente aranjate aleatoriu, pivotul este ales ca fiind primul:

```
def partitie(x,s,d):#x- tabloul, s- primul element al tabloului, d- ultimul element
        pivot=x[s]
        i=s-1
        j=d+1
                ...
        return j
```

Iar pentru listele în care valorile erau aranjate descrescător sau erau aproape sortate, dat fiind faptul că implementarea algoritmului a fost una recursivă, valoarea pivotului a fost aleasă ca elementul de la mijlocul tablolui ($m = \frac{n}{2}$):

```
def partitie(x,s,d):
        pivot=x[(d-s)/2]
        i=s-1
        j=d+1
                . . .
        return j
```

Datele se explică astfel: timpul de execuție a unui algoritm este datorat de numărul de operații efectuate - în dependență de tehnica folosită, parcurgerea efectivă a tabloului de câteva ori, sau împărțirea problemei în subprobleme. Pentru primii algoritmi caracteristic le este fie numărul de comparări, fie interschimbări , în dependență de principiul algoritmului.

[2]Implementările algoritmilor au fost făcute în Python. Acest fapt este menționat pentru că Python este un limbaj mai înalt decât C; C este un limbaj mai aproape de cod mașină și necesită mult mai puține prelucrări înainte de a fi procesat de calculator.
De aceea, timpii de execuție pot să difere în comparație cu alte limbaje de programare.

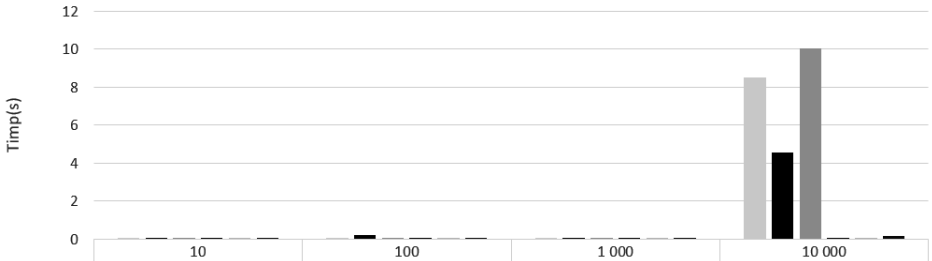
4 Studiu de caz

În această secțiune, urmează să tratăm un exemplu mai puțin trivial: sortarea unui tablou de câteva milioane de elemente. Dar până atunci, vom analiza fiecare metodă menționată mai sus, începând de la câteva zeci de elemente, până la zeci sau sute de milioane de elemente, în toate cele 3 ipostaze posibile de aranjare:

- aranjate aleatoriu,
- aproape sortate,
- sortate descrescător.

Pentru început, datele de intrare au fost numere generate înafara funcției de bază, fie printr-un algoritm de randomizare, fie ținând cont de tipul tabloului(aproape/desprescator sortat).
Testele au fost efectuate de 10 ori pentru o mai bună si mai acurată detaliere a timpului de execuție.

Figura1 prezintă o comparație generală a performanțelor algoritmilor menționați pentru tablouri de până la 10 mii de elemente. Astfel, prin acesta, se subliniază încă o data caracterul și importanța complexității algoritmilor.



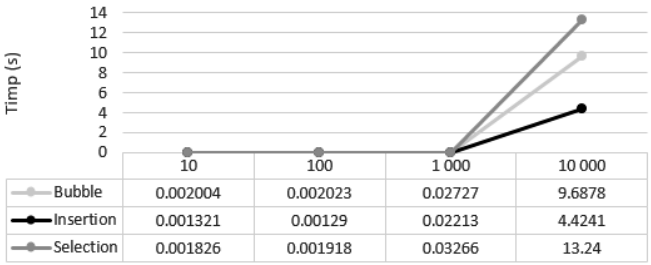
	10	100	1 000	10 000
Bubble	0.001	0.005	0.061	8.51
Insertion	0.0005	0.2023	0.003	4.55
Selection	0.0007	0.004	0.06	10.05
Quick	0.001	0.001	0.006	0.018
Count	0.000078	0.000096	0.001	0.005
Merge	0.0007	0.0076	0.018	0.15

Figura 1: Comparație de ansamblu algoritmi de sortare

Iar în următoarea parte, vor fi tratate cele 3 cazuri de date aranjate în tablou.

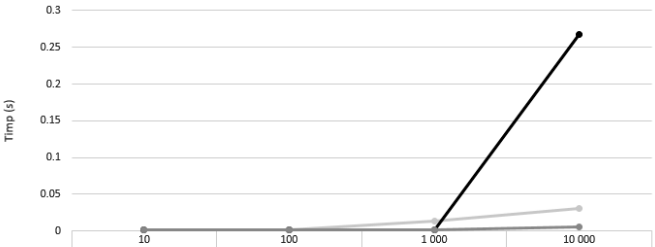
Caz I: Elemente aranjate aleatoriu

În acest caz, performanțele fiecărui algoritm au atins așteptările teoretice. Astfel, Figura 2 subliniază încă o dată importanța construcției algoritmului, timpului de execuție și complexității sale. În plus, o observație ce urmează a fi folosită în următoarele cazuri este că algoritmul de sortare



	10	100	1 000	10 000
Bubble	0.002004	0.002023	0.02727	9.6878
Insertion	0.001321	0.00129	0.02213	4.4241
Selection	0.001826	0.001918	0.03266	13.24

(a) BubbleSort, Insertion Sort, Selection Sort



	10	100	1 000	10 000
Quick	0.001272	0.001278	0.01318	0.03
Merge	0.001105	0.001928	0.002196	0.2674
Count	0.0013896	0.002	0.001	0.005

(b) QuickSort, MergeSort, CountingSort

Figura 2: Timpi de execuție elemente aleatorii

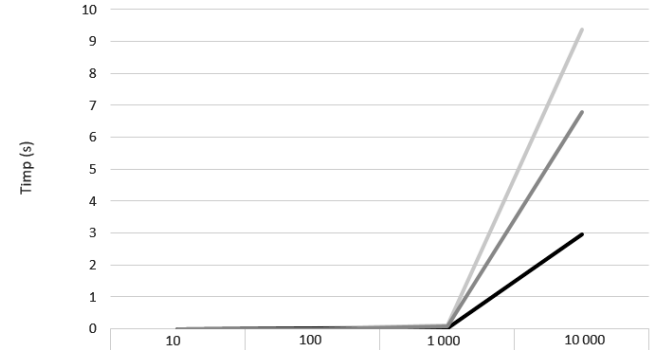
prin inserție Figura 2a este cel mai eficient, în comparație cu semenii săi cu complexitatea de $O(n^2)$. Până acum, nu apare nimic nou.

Caz II: Elemente aproape sortate

Conform rezultatelor precedente, Insertion Sort a demonstrat prin performanțele sale că este un algoritm mai eficient pentru liste aproape sortate.Deci, dat fiind faptul ca Quicksort este un algoritm bun în special pentru liste mari, în acest caz, pe lângă algoritmul clasic, am implementat un algoritm hibrid, în care, dacă lungimea listei era mai mică de o constantă X, acesta urma să aranjeze elementele folosind sortarea prin inserție. În urma experimentului efectuat, am înregistrat o scădere a timpului de execuție a programului cu $2 - 8\%$. Această scădere a fost sesizată atunci când listele urmate a fi sortate erau de 100 000 de elemente, iar în momentul, când se divizau în subprobleme a căror lungime era de 400-500 elemente, acestea se sortau cu Insertion Sort1.

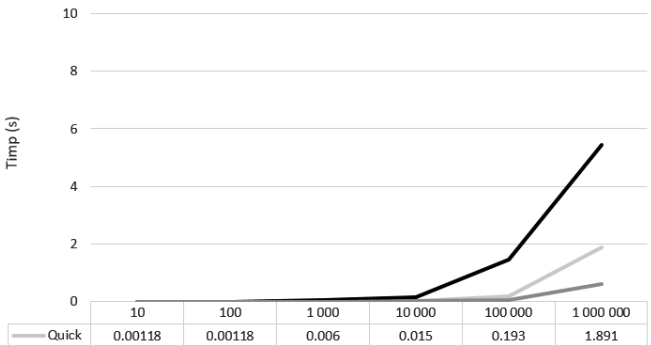
Algoritm 1: Quicksort hibrid

```
def Quick(x,s,d):
        if len(x)<500:
                return insertion(x)
        if s<d:
                q=partitie(x,s,d)
                x=Quick(x,s,q)
                x=Quick(x,q+1,d)
        return x
```



	10	100	1 000	10 000
Bubble	0.00047	0.005	0.093	9.37
Insertion	0.00057	0.0062	0.029	2.97
Selection	0.000003	0.00113	0.069	6.8

(a) BubbleSort, Insertion Sort, Selection Sort



	10	100	1 000	10 000	100 000	1 000 000
Quick	0.00118	0.00118	0.006	0.015	0.193	1.891
Merge	0.0012	0.000173	0.05	0.147	1.443	5.442
Count	0.00018	0.00007	0.001	0.006	0.064	0.619

(b) QuickSort, MergeSort, CountingSort

Figura 3: Timpi de execuție elemente aproape sortate

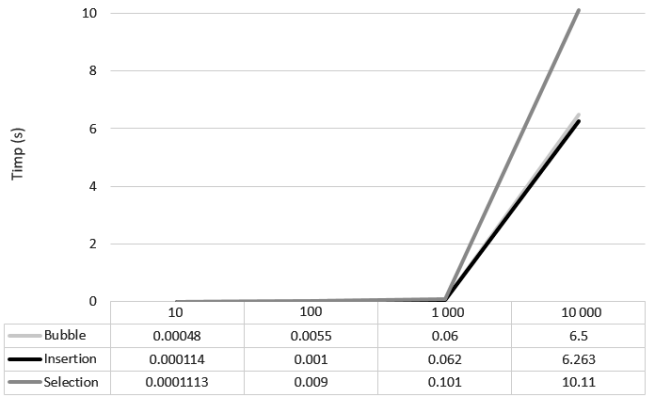
Desigur că există tot felul de algoritmi care încearcă să îmbine avantajele unora, fără a lua și dezavantajele cu ei. Unii algoritmi care folosesc conceptul altor:

- HeapSort, SmoothSort, Weak-heap sort - Selection Sort,
- ShellSort, SplaySort, TreeSort - Insertion Sort,
- etc.

Ce ține de rezultatele înregistrate de fiecare algoritm în parte, se observă aceeași tendință de scădere a timpului necesar, mai puțin la BubbleSort, care este nevoit să facă toate comparările pentru a ajunge la final. Și QuickSort,MergeSort, CountingSort s-au descurcat mai bine cu aceste date. Ceea ce demonstrează încă o dată că eficiența și performanța unui algoritm ține și de felul aranjării datelor în listă.

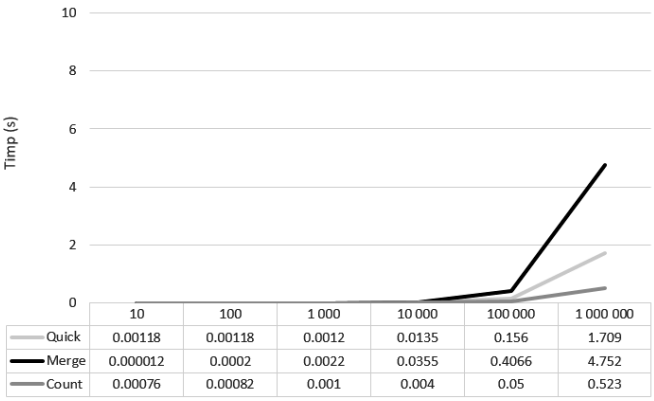
Caz III: Elemente aranjate descrescător

Cazul cel mai nefavorabil, care scoate tot ce-i mai urât de oricare algoritm de sortare.



	10	100	1 000	10 000
Bubble	0.00048	0.0055	0.06	6.5
Insertion	0.000114	0.001	0.062	6.263
Selection	0.0001113	0.009	0.101	10.11

(a) BubbleSort, Insertion Sort, Selection Sort



	10	100	1 000	10 000	100 000	1 000 000
Quick	0.00118	0.00118	0.0012	0.0135	0.156	1.709
Merge	0.000012	0.0002	0.0022	0.0355	0.4066	4.752
Count	0.00076	0.00082	0.001	0.004	0.05	0.523

(b) QuickSort, MergeSort, CountingSort

Figura 4: Timpi de execuție elemente sortate descrescător

Și în această parte am încercat hibridizarea celor 2 algoritmi - QuickSort și Insertion Sort. Am făcut analiza pe liste de 10 000 elemente, astfel încât, după divizarea problemei până la una de lungimea egală cu 50, se efectua sortarea cu algoritmul de sortare prin inserție. Aici însă, timpul s-a redus considerabil cu circa $10 - 12\%$, față de 8% la listele cu elemente aproape sortate.

Cum am menționat anterior, limbajul de programare are o influență care trebuie menționată asupra timpilor de execuție.

Așadar, un alt factor care acționează asupra lor este programarea paralelă. Programarea paralelă este metoda prin care diferite sarcini sunt îndeplinite de diferite procesoare în același timp.

În concluzie, fiecare metodă folosită își are avantajele și dezavantajele proprii. Unele dezavantaje se dovedesc a fi mai performante decât unele avantaje. Iar acest fapt simplifică alegerea atunci când se cunoaște cum sunt aranjate datele înainte de a fi procesate dar și numărul acestora.

5 Comparaţie cu litereatura

Această lucrare încearcă să prezinte o comparație practică a performanțelor de timp a unor algoritmi de sortare uzuali. Algoritmii propuși de **John Von Neumann [Neu]** și **Tony Hoare [Hoa]** au fost implementați, iar mai apoi s-a încercat o îmbunătățire a celui din urmă cu ajutorul lui Insertion Sort.

Multe lucrări din domeniu abordează algoritmii de sortare, dat fiind faptul că aceștia stau la baza multor aspecte ale industriei. De obicei însă, autori importanți și operele lor precum Steven Skiena în [Ski], Donald E. Knuth în [Knu] dar și Thomas Cormen în [CLRS09] prezintă o analiză în general teoretică a acestora, explică principiile algoritmilor, complexitatea dar și alte detalii.

Lucrări care au drept scop analiza și merită a fi menționate sunt puține, printre acestea este însă *A Survey, Discussion and Comparison of Sorting Algorithms*[1], o teză care prezintă o mai detaliată a câtorva concepte de bază începând cu analiza complexității unui algoritm până la exemple de sortări a liste de numere dar și șiruri de caractere.

Pe piață insă, există algoritmi de sortare care încearcă să remedieze - QuickSort 3, HeapSort. Însă au și ei dezavantajul unui Worst Case[2] de $O(n^2)$ sau $O(nlog(n))$.

Oamenii de știință încearcă să scoată cei mai bun din algoritmii deja existenți. Astfel, se observă multe versiuni, asa numite *Enhanced* (îmbunătățite), ale algoritmilor de bază în care complexitatea rămâne aceeași, însă performanța crește în urma efectuării altfel a pașilor sau a implementării diferite.

Un avantaj al acestei lucrări, în comparație cu lucrările ce studiază partea teoretică a algoritmilor, este faptul că tratează cei mai importanți 6 algoritmi folosiți uzual de către programatori; iar pe lângă aceasta, mai face o comparație practică a rapidității acestora, cu ajutorul unor diagrame. În plus, este de remarcat faptul că s-a încercat o dezvoltare a unui algoritm care îmbină dinamismul și rapiditatea lui *QuickSort* pe tablouri mici cu performanța lui *Insertion sort* pe tablouri mici, atât pe liste aproape ordonate, cât și pe cele aranjate descrescător.

Pe de altă parte, un dezavantaj major este impotența studierii a cât mai multor algoritmi prezenți pe piață pentru o gamă mai largă de rezultate și o comparație mai efectivă, dar și lipsa efectivă a mult mai multor detalii matematice în legătură cu formulele matematice, formule de complexitate ș.a.

[1] Lucrarea a fost prezentată în cadrul departamentului de informatică al Universității Umeå din Suedia, în 2014

[2] Prin *Worst case scenario* se înțelege resursele maxime necesare cerute în cazul cel mai nefavorabil.

6 Concluzii şi direcţii viitoare

Această lucrare prezintă o comparație teoretică și practică a performanțelor algoritmilor de sortare dar și rezultatele aferente. Principalele constatări ale acestei lucrări sunt că numărul de operații efectuate, timpul de execuție și complexitatea algoritmului sunt factori determiniști când vine vorba de o comparare a algoritmilor de sortare și de găsirea unui algoritm optim. Totuși, pe lângă așteptările care s-au adeverit, au apărut și hint-uri care au dus la combinarea unor algoritmi pentru o mai bună performanță în definitiv.

În urma efectuării sortării de tablouri pentru fiecare algoritm menționat mai sus, rezultatele timpurilor demonstrează încă o dată: complexitatea își spune cuvântul când vine vorba de un număr mare de elemente. Pentru sortarea unui număr mic de elemente, toate metodele s-au descurcat în timp minim(< 1 s). Abia când are loc trecerea la tablouri cu 10.000 până la 10.000.000 elemente, atunci fiecare metodă își spune cuvântul, prin rapiditate și timpi de execuție specifici.

Secțiunea 4. Studiu de caz confirmă aspectele și așteptările propuse la început ce țin de timpul de execuție a fiecărui algoritm de sortare a datelor; însă pe lângă aceasta, am identificat un aspect cel puțin interesant ce merită menționat: combinația de algoritmi dintre QuickSort și Insertion sort.

Problema timpului mare execuție în programarea simplă a dus la apariția unor ramuri noi ale informaticii - *Programarea distribuită , Programarea paralelă.* care duc la folosirea puterii de calcul a tuturor componentelor calculatorului pentru atingerea aceluiași scop - rezolvarea în comun a aceleași probleme.

O așteptare care nu s-a îndeplinit este faptul că o listă cu peste 100 mii elemente nu a putut fi sortată cu un algoritm de complexitate $O(n^2)$ precum *BubbleSort, Insertion sort, Selection sort.* Acest fapt are drept cauză numărul imens de operații necesare pentru procesarea unui astfel de tablou.

În final, această lucrare lasă câteva întrebări deschise care urmează ulterior a le fi găsit răspunsul dacă există, iar dacă nu, va rămâne deschisă până la apariția unei soluții.

În fond, care e cel mai eficace și optim algoritm de pe piață atunci când nu se cunosc datele de intrare, felul dar și natura lor?
Pe lângă programarea distribuită și paralelă, cum putem optimiza algoritmii de sortare?
Pot structurile de date dinamice să îmbunătățească performanța acestor algoritmi?

Bibliografie

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms(Third Edition)*. 2009.

[Hoa] C.A.R. Hoare. Algorithm 64, quicksort. 4:321.

[Knu] Donald E. Knuth. *The art of computer programming Vol3*. 2 edition.

[Neu] John Von Neumann. The computer as von neumann planned it. 15(1).

[Ski] Steven S. Skiena. *The Algorithm Design Manual, 2nd edition*.

[Zah] D. Zaharie. *Introducere în proiectarea și analiza algoritmilor*.